

## MASSIVELY REDUCED INSTRUCTION SET PROCESSOR

### FIELD OF INVENTION

This invention relates in general to microprocessors, and in particular, to a  
5 microprocessor used for data communications.

### BACKGROUND OF THE INVENTION

Over the last few decades Internet Protocol (IP) communications have become the  
dominant form of electronic communication. IP communications allow the use of a wide  
10 array of different protocols. To simplify data handling and routing, the protocols are  
arranged in a stack and the “lowest-level” protocols encapsulate the higher-level  
protocols. This encapsulation allows the idiosyncrasies of the higher level protocols to be  
hidden from the routing functions and further allows the partitioning of the analysis of  
the data.

15 In stand-alone devices, also known as embedded products and embedded devices,  
embedded computers are typically used to perform the encapsulation and de-  
encapsulation to send and receive the data respectively. An embedded computer is  
characterized as having a general purpose CPU, with associated memory. The computer  
20 runs an Operating System (OS), such as embedded Linux. The protocol processing is  
handled by the OS and application software is provided that runs on top of the OS to  
handle the communications functions and other tasks that are required.

25 This architecture is analogous to what is provided on general purpose computers (PCs)  
and workstations. Using the same processes to handle the communications in the  
embedded device as are used on general purpose computers is natural since IP  
communications was first performed only on general purpose computers and later  
migrated to embedded devices.

30 However, different from general purpose computers, embedded devices only have  
limited resources and are highly cost sensitive. The processor that can be employed in an  
embedded computer is often very limited in performance due to cost, space, and power  
consumption constraints. As a result an embedded device often cannot be cost effectively  
IP enabled for high-bandwidth devices.

To handle multiple tasks a real-time operating system (RTOS) is often employed which provides the abilities to respond to system requests in a very short period of time. Even with this, applications such as high performance image delivery for machine vision find 5 the level of latency and the variation in the latency associated with the delivery of the video to be unacceptable. Further, when OS-based embedded devices are pushed to their limits they can become unreliable with deadlocks that freeze the device.

It is obvious that the above implementations do not address the requirements for protocol 10 processing on a device, such as a high-speed electronic video camera or other high-bandwidth device. Therefore there is a need for a method and apparatus capable of processing IP packets with low, consistent latencies that are suitable for delivering video over an IP network.

15 **SUMMARY OF THE INVENTION**

This invention is directed to a method and apparatus for providing low, predictable latencies in processing IP packets. The apparatus provides a specialized microprocessor or hardwired circuitry to process IP packets for video communications and control of the video source without an operating system. The method relates to operation of a 20 microprocessor which is suitably arranged to carry out the steps of the method. The method includes details of operation of the specialized microprocessor.

In accordance with one aspect of this invention, a massively reduced instruction set processor (mRISP) is disclosed which is a tiny embedded soft processor tailored for 25 processing communication protocols in accordance with the method disclosed herein. In a preferred embodiment, this processor has only two instructions and some optional registers performing basic functions, such as arithmetic and logical functions, and specialized functions like Program Counter, Timers, IP Checksum and DMA (Direct Memory Access). The soft implementation of the mRISP is realized since it is fully 30 configurable upon construction through synthesis of a register transfer level (RTL) representation of the design by specifying the registers and the features required in the implementation. The processor that is created from the synthesis is tailored for a specialized task, such as data communications.

The two mRISP instructions are LOAD and MOVE which are the minimal instructions necessary for a processor. Some macros are built over these two instructions in conjunction with registers to add some other basic functionality like JMP, CALL and

5 RET. The macros are used in the compiler for the instruction set for the mRISP, and are built solely using the LOAD and MOVE instructions.

The core is maximally optimized for a 16-bit data bus and a 32-bit instructions bus, although it can be configured for wider or narrower bus widths. In 16-bit data mode,

10 bytes can be swapped for single byte access and operation. The 32-bit instructions bus, separated from the data bus, allows the timing to be reduced to only one clock cycle for a LOAD and two clock cycles for a MOVE. An extra clock cycle is added to the timing on a jump in the program counter.

15 For slow external memory fetching or for any other specific reasons, external logic can be added to control the HOLD input signal and holds the processor for a required number of clock cycles. In addition to that, specialized waiting functions, if required and activated, can hold the processor until an expected event occurs.

20 With such a processor, IP packets can be processed at significantly higher rates, with lower, consistent latencies, than can be accomplished using a general purpose microprocessor.

## **BRIEF DESCRIPTION OF THE DRAWINGS**

25

Figure 1 is a functional block diagram of a microprocessor constructed in accordance with the preferred embodiment of the invention;

30 Figure 2 is a state diagram of the mRISP State Machine element of the microprocessor of Figure 1;

Figure 3 is a functional block diagram of the Data Path of the User Memory/registers portion of Figure 1.

Figure 4 is a functional block diagram of an embodiment a CSUM checksum register.

5

Figure 5 is a layout of a MOVE instruction.

Figure 6 is a layout a LOAD instruction.

10 Figure 7 is a field layout and description of a general purpose register, REG\_A.

Figure 8 is a field layout and description of a second general purpose register, REG\_B.

Figure 9 is a field layout and description of a program counter register, PCNT.

15

Figure 10 is a field layout and description of a return register, RETA.

Figure 11 is a field layout and description of a mask register, MASK.

20 Figure 12 is a field layout and description of a wait register, WAIT.

Figure 13 is a field layout and description of a timer register, TIMER0.

Figure 14 is a field layout and description of another timer register, TIMER1.

25

Figure 15 is a field layout and description of a checksum register, CSUM.

Figure 16 is a field layout and description of a direct memory address register, DMA.

30 Figure 17 is a table listing jump and call conditions when writing in the Program Counter.

Figure 18 is a list of a possible set of macros that could be used to implement jump and call directives in a two op code microprocessor constructed in accordance with the principles of the invention.

5 Figure 19 is a functional block diagram of a preferred implementation of the instruction formatter with the opcode decoder elements of Figure 1 that include an address detector and a byte swapping detector.

10 Figure 20 is a functional block diagram of a preferred embodiment of the Internal Registers and Functions elements of Figure 1.

Figure 21 is a functional block diagram of a preferred implementation of the Event Block element of Figure 20.

15

#### **DETAILED DESCRIPTION OF THE PREFERRED EMBODIMENT**

In the following detailed description of the embodiments, reference is made to the accompanying drawings, which form a part hereof, and in which is shown by way of illustration specific embodiments in which the invention may be practiced. These embodiments are described in sufficient detail to enable those skilled in the art to practice the invention, and it is to be understood that other embodiments may be utilized and that structural, logical and electrical changes may be made without departing from the spirit and scope of the present inventions as defined in the claims appended hereto.

20 25 The following detailed description is, therefore, not to be taken in a limiting sense, and the scope of the present inventions is defined only by the appended claims.

30 Figures 1 and 22 depict a preferred embodiment of the massively reduced instruction set processor (mRISP) constructed in accordance with the principles of the invention. The mRISP implements the CPU with separate data and program memory bus, generally known as a Harvard memory architecture. Preferably, the mRISP program memory bus is a 32-bit wide bus that is used to fetch instructions from memory. The mRISP data bus

is preferably a 16-bit wide bus that is used to move 16-bit word data from user memory, internal registers or program memory to user memory and internal registers. The external user memory bus may be connected to memories or peripherals. It is through these connections that communication and control data is transferred to and from the mRISP 5 for processing and analysis.

For example, in a camera capable of high-speed IP communications, the video data will be placed in user memory through an external controller. The mRISP can be signalled via irq[3:0] signals, as depicted in Figure 1, to prepare the IP headers for 10 communications and finalize the packet. Similarly, once the packet is finalized, a write in a dedicated peripheral port (which can be a register) is used to signal the communications interface to send the data. The inverse of this method is used when receiving a packet. The external controller fills the user memory with the incoming packet and signals the mRISP through irq[3:0] signals. Then the mRISP processes the packet in its user 15 memory and take the appropriate action depending on the contents of the packet. Control and configuration information is transferred in an identical manner.

### Instruction Set

20 The mRISP instruction set is massively reduced to only two instructions. The first one is the instruction MOVE which one moves data from a source address to a destination address. The only other instruction necessary for a functional CPU is the instruction LOAD, which one can initialize memory and registers to a proper value from the program memory.

25 Therefore, in the preferred implementation each 32-bit instruction contains only one bit to encode the opcode. On an instruction MOVE, 14-bits are dedicated for the source address and another 14-bits are used for the destination address, leaving 3 bits unused. On an instruction LOAD, 14-bits are used for the destination address and 16-bits for the 30 constant word to load, leaving 1 bit unused.

The most significant bit (MSB) bit of the addresses (source and destination) is used to select between the external user memory region and the internal registers region. The

least significant bit (LSB) bit of the addresses (source and destination) is used to decode if data bytes swapping has to be done. Thus 12 bits out of 14 bits are available to user memory and peripheral. The external memory address is in word (16-bit).

5 Figures 5 and 6 provide layout giving a bit by bit description of the MOVE and LOAD instructions.

Figure 19 provides an implementation of the instruction formatter with the opcode decoder, address detector and byte swapping detector.

10

#### Internal Registers and Functions

Preferably, the mRISP has two general-purpose registers (REG\_A and REG\_B), the layout of which are shown in Figures 7 and 8. The mRISP preferably also has some

15 dedicated registers performing specific functions (i.e. features). Those functions may be any combination or number of arithmetic (increment, adder), logical (AND, OR, XOR), comparators, timers see Figure 14, DMA see Figure 16, interrupts as shown in Figure 10 and program counter as shown in Figure 9 functions. The arithmetic and logical registers use the general-purpose registers with the mask register as shown in Figure 11 as inputs.

20 Their values are constantly updated as general-purpose registers change.

Two registers are specifically designed to process Internet Protocols. The first one (CSUM) which is shown in Figure 4 and Figure 15. The CSUM register is useful to compute Internet Protocols checksums. The method used to compute the IP checksum is the 16-bit one's complement sum of the corresponding data. Each time a write is done into the CSUM register, the 16-bit one's complement addition is computed from the previous value and the written value. When all the data to be included in the checksum has been written in this register, the read of this register gives the 16-bit one's complement sum by inverting the present value. A read resets the CSUM register to zero,

25 ready for another computation. By filling the checksum field(s) in the IP header(s) with a magic number, the checksum can be serially performed as the data is being packetized. One word is added at the end of the packet with the appropriate data necessary to make the magic number in the header field correct.

The second register is the DMA register which is shown in Figure 16. The (DMA) register is used to move multiple data from one location to another one within three instructions. When one location is an internal register, its address is not incremented, enabling the capability to send consecutive data in memory into one special register or initialize consecutive data in memory with one register's value. In conjunction with the CSUM register, the DMA register is used to quickly and easily compute Internet Protocols checksums with only a few instructions.

- 5      Comparisons between REG\_A and REG\_B are constantly computed. Two flags are necessary to denote all possible comparison outcomes, namely, equal '==', not equal '!=', less than '<', greater than '>', less than or equal '<=' and greater than or equal '>='. The first flag of the two flags is the "A Equal B" flag (eq) and the second one is the "A Greater than B" flag (gt). Those flags are used in conjunction with the Program Counter (PCNT) to enable conditional jumps. The descriptions of the General Purpose registers, Program Counter, Return Register, Mask Register, Wait Register, Timer 0 Register, Timer 1 Register, Checksum Register and DMA Register are provided in Figures 7 – 15 respectively.
- 10     Figure 20 is a functional block diagram of a preferred embodiment of the Internal Registers and Functions elements of Figure 1. Figure 20 shows the interoperation of the Internal Registers and Functions.
- 15     Figure 21 is a functional block diagram of a preferred embodiment of the Internal Registers and Functions elements of Figure 1. Figure 21 shows the interoperation of the Internal Registers and Functions.
- 20     Figure 22 is a functional block diagram of a preferred embodiment of the Internal Registers and Functions elements of Figure 1. Figure 22 shows the interoperation of the Internal Registers and Functions.

#### Program Counter and Return Registers

- 25     The Program Counter register (PCNT) is cleared to zero on reset and is incremented by one on the last cycle of every instruction (when *prd*, see Figure 1, is high). The PCNT always points to the next instruction during the processing of the current instruction. A jump in the program memory can be accomplished by writing the new instruction's address in the Program Counter register. The jump can be conditional or not, depending on the state of the comparator flags (eq and gt) and the setting of the three flag bits (IE, IG and IN) in the Program Counter Register.
- 30     Figure 23 is a functional block diagram of a preferred embodiment of the Internal Registers and Functions elements of Figure 1. Figure 23 shows the interoperation of the Internal Registers and Functions.

Figure 17 summarizes the jump and the call conditions when writing in the Program Counter.

A CALL instruction can be accomplished by writing in the Program Counter register the 5 sub-routine's address and by setting the flags to IE=0, IG=0 and IN=1. In this case, the Return register (RETA) loads the Program Counter's value at the same time the jump is done. Later, on a RET instruction (by moving RETA's value into PCNT register), the mRISP can resume fetching instructions on the next one's after the CALL instruction. The stack is hardware and its depth is configurable at the synthesis. The stack is 10 structured as a LIFO (Last In First Out). On a CALL instruction, the Program Counter's value is pushed in the LIFO and on a RET instruction, the value to write into the Program Counter is pulled from the LIFO.

## 15 Event Handling

In the preferred implementation the mRISP allows up to 16 events, which can be generated from any of the two sources: external hardware interrupts or internal events. The internal events may come from timers, real-time timer and watchdog logic. All 20 events are completely handled by software (or firmware) and no event can interrupt the execution of the program. The software must verify itself in the WAIT register if an event occurred. The software can put the processor in the sleep mode by setting in the WAIT register the bit(s) of the corresponding event(s) it want to be waked up.

25 According to Figure 12 and Figure 21, writing one in the event 'X' bit of the WAIT register, sets to one the corresponding "SET" signal (*wait\_x\_set*) and, at the same time, sets to one the global signal *wait*. Then the processor goes in the sleep mode and waits for the event X.

30 When this event occurs (*event\_x* goes to one), the corresponding "EVENT" signal (*wait\_x\_evt*) is set to one. One clock cycle later, this signal clears the SET signal (*wait\_x\_set*) and the global signal *wait*. Thus the processor resumes its operations.

The software has the responsibility to clear the EVENT bit and to retrieve which event waked up the processor if more than one bit has been set in the wait register. By reading the WAIT register, the software reads all the EVENT bits (*wait\_?\_evt*) and also clears most of the bits (timer event bits are only clear by writing in the corresponding TIMER register).

5 Figure 21 provides an implementation of the Event Block.

### Macros

10

Macros are added to instructions that are interpreted by the compiler. These make the mRISP easier to program and makes the resulting assembly code more understandable and maintainable. These are built over the two instructions in conjunction with registers. For example the JMP macro, which one is used to jump in another part of the program, is 15 in fact a LOAD instruction with the destination address equals to the Program Counter register's address and the constant data equals to the address to jump in the program memory.

20 Figure 18 provides a listing of a possible set of macros that could be used.

20

### Data Path

For each instruction, a 16-bit data word is transferred from one location to another one. The source may be from the program memory (on a LOAD), from one of the internal 25 registers or from the user memory (on a MOVE). The destination may be either one of the internal registers or the user memory.

The higher byte and the lower byte in the data may be swapped together when only one of the location address is odd (bit 0 is high). This is very useful to reverse the byte 30 ordering since Internet Protocols are big-endian and the mRISP is little-endian.

Figure 3 is a functional block diagram of the data path used by the microprocessor of the present invention.

### State Machine

5 The mRISP state machine synchronizes internal and external control signals to provide efficient timing. The LOAD instruction takes only one clock cycle and two clock cycles are taken to execute the MOVE instruction. An extra clock cycle is added to the timing on a jump in the program counter.

10 Figure 2 provides a state diagram of the mRISP State Machine portion of Figure 1. A mRISP constructed in accordance with the present invention, has only four states RESET, FETCH32, WAIT\_ACK and JUMP. The RESET state is reached whenever the signal *rst\_n* is asserted on the so named signal line shown in Figure 1. At the first cycle where *rst\_n* is de-asserted, the state machine goes to the FETCH32 state.

15 The FETCH32 state decodes the instruction presented on the *pdata\_in* bus. Depending on the value of the opcode and the signals *hold* and *wait* on the so named signal lines shown in Figure 1, the next state can be WAIT\_ACK, JUMP or FETCH32 again. The signal *wait* is used in this state to keep the processor waiting for an event, defined previously by writing in the WAIT register. During this waiting, no instruction fetching,  
20 no writes and no reads are performed. In the FETCH32 state, the signal *hold* has the same effect as the signal *wait* but it is generated by external logic. The reason for its assertion may be that data from the program memory is not ready due to slow memory, that the write from the previous instruction into external memories takes more than one clock cycle or for any other reasons. If the signals *wait* and *hold* are not asserted and the  
25 opcode is MOVE, a read is performed from the source address and the next state is WAIT\_ACK. Otherwise the instruction LOAD is performed. The constant data contained in the instruction is written to the destination address. If the destination address is the Program Counter Register (PCNT) and the flag indicates an unconditional jump or a true conditional jump (signal *jump* is asserted), the State Machine goes in the  
30 JUMP state. Otherwise, it stays in the same state, ready for the next instruction.

The WAIT\_ACK state waits for the read data from the source address to be ready. If it's not, the external logic must keep the signal *hold* asserted until data is ready. When it is

ready, the State Machine comes back in the FETCH32 state unless the destination address of the MOVE instruction was the Program Counter Register (PCNT) and the flag indicated an unconditional jump or a true conditional jump (signal *jump* is asserted). In this last case, the next state is going to be JUMP.

5

The JUMP state is an idle state where the cycle is used only to fetch the instruction pointed by the new address loaded in the Program Counter Register.

The State Machine comes back in the FECTH32 state unless the external logic keeps the signal *hold* asserted for any reason.

10

One skilled in the art will recognize that this description provides a very lean implementation of a processor optimized for handling the delivering data from one location to another. Through this lean implementation data can be moved very quickly and efficiently providing for high efficiency and low latency in data transfer.

15

Through the software for the mRISP, IP packets can be generated and moved with the associated data payload. Since the processor provides a single thread of processing, and the use of an operating system is not required, latencies are predictable. It will be noted that an operating system is undesirable since the mRISP is best suited to perform a single 20 function, such as IP packet processing, and an operating system would increase the latency and reduce the predictability of the system.

It is to be understood that this description is intended to be illustrative, and not restrictive. Many other embodiments will be apparent to those of skill in the art upon 25 reviewing the above description. The scope of the invention should, therefore, be determined with reference to the appended claims, along with the full scope of equivalents to which such claims are entitled.

The embodiment(s) of the invention described above is (are) intended to be exemplary 30 only. The scope of the invention is therefore intended to be limited solely by the scope of the appended claims.

## **I CLAIM**